MD5 Collision Attack Writeup

Liam Ryan

Contents

Contents

Contents 1

Introduction 1

The objective of this lab is to gain insight into the importance of the collision-resistance property of a hash. For a one-way hash function to be considered secure there are several properties it must satisfy. The first is the one-way property; consider *hash(M) = h.* 1

Environment 1

Performed inside an Ubuntu VM (20.04) using the md5collgen program written by Marc Stevens. 1

Task 1: Generating Two Different Files with the Same MD5 Hash 2

Task 2: Further Understanding MD5's Property 6

Task 3: Generating Two Executable Files with the Same MD5 Hash 7

Task 4: Making the Two Programs Behave Differently 11

Citations 16

Introduction

The objective of this lab is to gain insight into the importance of the collision-resistance property of a hash. For a one-way hash function to be considered secure there are several properties it must satisfy. The first is the one-way property; consider *hash(M) = h.*
Given *h* it must be infeasible to find the input M.

The second one and the focus of this lab is the collision-resistance property. It must be computationally infeasible (cost more to do than the information is worth) to find the hash of two different inputs equal. A collision occurs when this property is violated. i.e.,

$Hash(M_1) == Hash(M_2)$

In this Lab we will take advantage of this collision property (or lack thereof) to create to two distinct programs that produce the same MD5 checksum (hash).

Environment

Performed inside an Ubuntu VM (20.04) using the md5collgen program written by Marc Stevens.

Task 1: Generating Two Different Files with the Same MD5 Hash

To start this lab the md5collgen program and a sample text file are required. We want to ensure the program works and can generate the same hash for two different text files before we proceed with the actual attack.

```
liam@ubuntu:~/Lab1$ touch lab1md5.txt
liam@ubuntu:~/Lab1$ ls
lab1md5.txt  md5collgen
liam@ubuntu:~/Lab1$ vi lab1md5.txt
liam@ubuntu:~/Lab1$
```

```
i am the contents of the file.
~
~
~
```

I then ran the program using the previously created text file.

```
liam@ubuntu:~/Lab1$ ./md5collgen -p lab1md5.txt -o out1.bin out2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1.bin' and 'out2.bin'
Using prefixfile: 'lab1md5.txt'
Using initial value: fb7b32aa3ca2612bb54b3900568725ec

Generating first block: .......
Generating second block: W.........................
Running time: 4.67827 s
liam@ubuntu:~/Lab1$
```

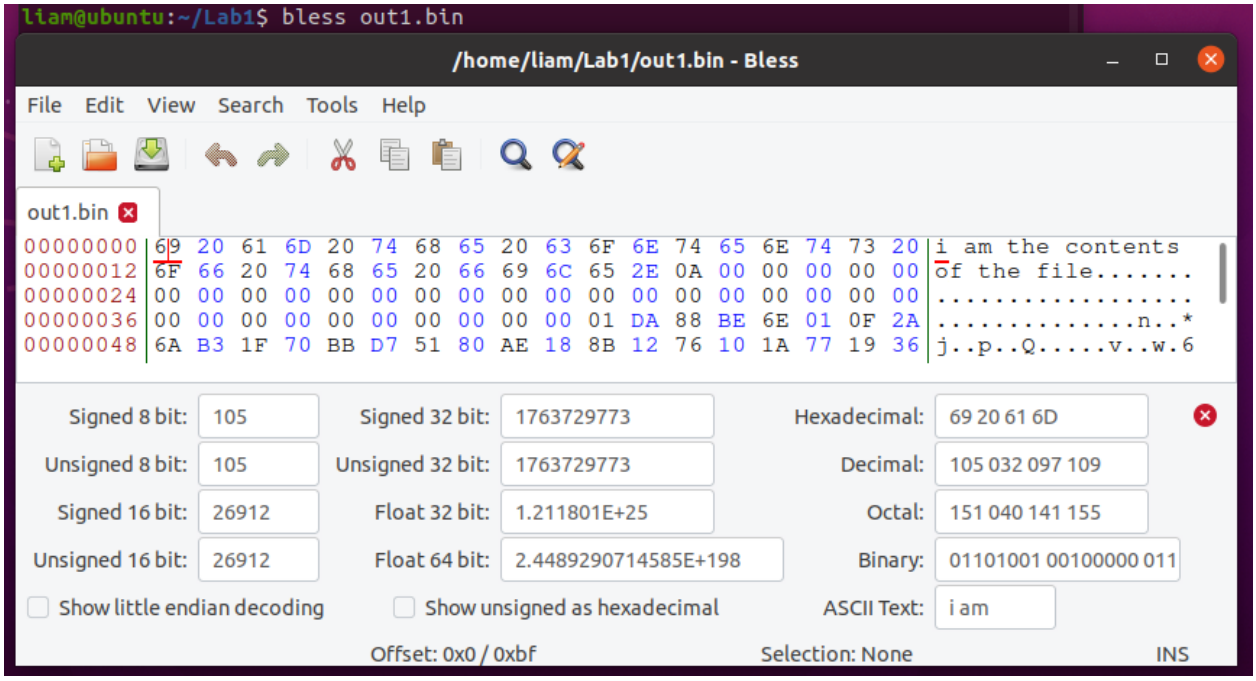
Using *diff* on the two output files we can determine if they are different, it determined that they are.

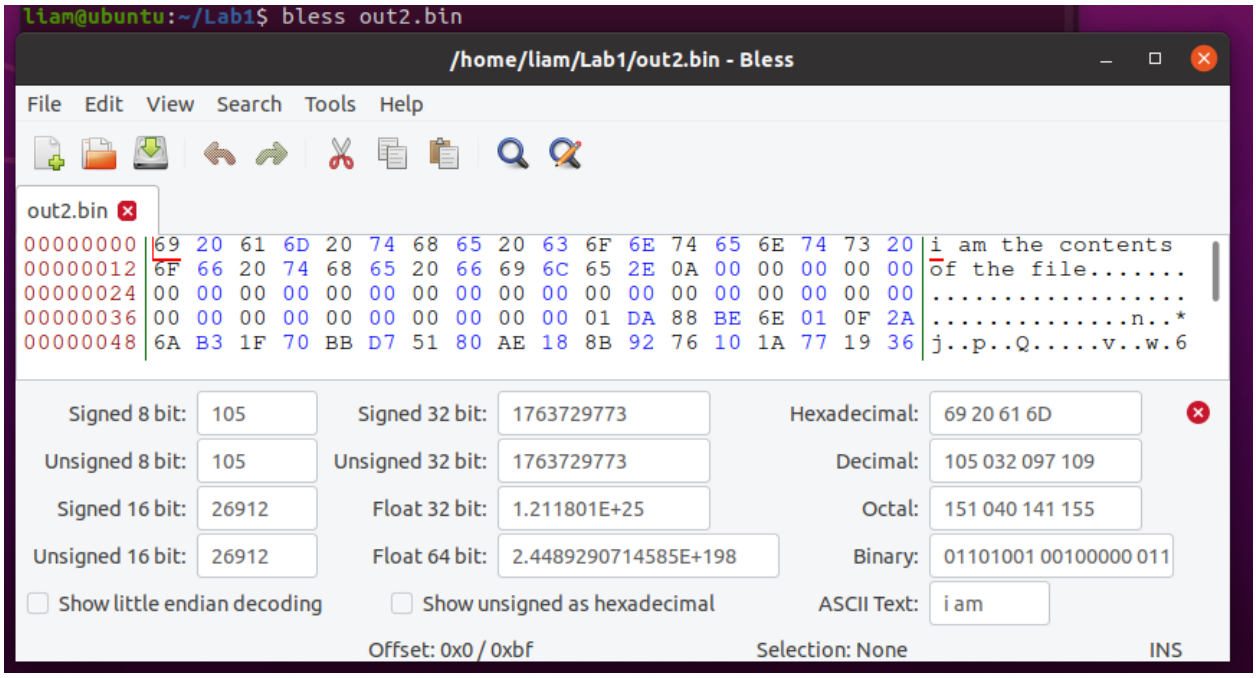
```
liam@ubuntu:~/Lab1$ diff out1.bin out2.bin
Binary files out1.bin and out2.bin differ
```

When performing a checksum on the two output files, despite them having an alternate 128-byte ending, they produce the exact same MD5 hash, meaning a collision has occurred.

```
liam@ubuntu:~/Lab1$ md5sum out1.bin
cbaacb8298be9dd80e128e5be0ede11a  out1.bin
liam@ubuntu:~/Lab1$ md5sum out2.bin
cbaacb8298be9dd80e128e5be0ede11a  out2.bin
liam@ubuntu:~/Lab1$
```

Using *bless* (a hex editor), I can dump the hex for both output files. The prefix appears on the right in plain text followed by 0 padding, and then what I assume is the appended P and some other data. The results of both dumps appear to be identical at first glance. But if you read through the hex, you can see the difference.





If the length of your prefix file is not multiple of 64, what is going to happen?

The remaining space gets padded with zeros; this is because MD5 encrypts 64-byte blocks.

Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.

Recreate the file with a prefix size of 64, and then ensure its size is 64.

```
liam@ubuntu:~/Lab1$ ls -l lab1md5.txt
-rw-rw-r-- 1 liam liam 64 Jan 12 12:57 lab1md5.txt
```

Re-run the MD5collgen program.

```
liam@ubuntu:~/Lab1$ ./md5collgen -p lab1md5.txt -o out1.bin out2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1.bin' and 'out2.bin'
Using prefixfile: 'lab1md5.txt'
Using initial value: b2291b48842948151ea655bc673edbf0

Generating first block: ............
Generating second block: S11...
Running time: 14.6277 s
```

I dump the hex again and can see that there is no zero padding.

out1.bin out2.bin

```
00000000 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00000012 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00000024 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00000036 41 41 41 41 41 41 41 41 0A 0A EF 03 79 C5 90 49 25 41 AAAAAAAA....y..I%A
```

out1.bin out2.bin

```
00000000 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00000012 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00000024 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00000036 41 41 41 41 41 41 41 41 0A 0A EF 03 79 C5 90 49 25 41 AAAAAAAA....y..I%A
00000048 42 C7 C4 AB 68 B3 C1 68 23 47 C0 58 BA 52 04 5F 2D 46 B...h..h#G.X.R._-F
```

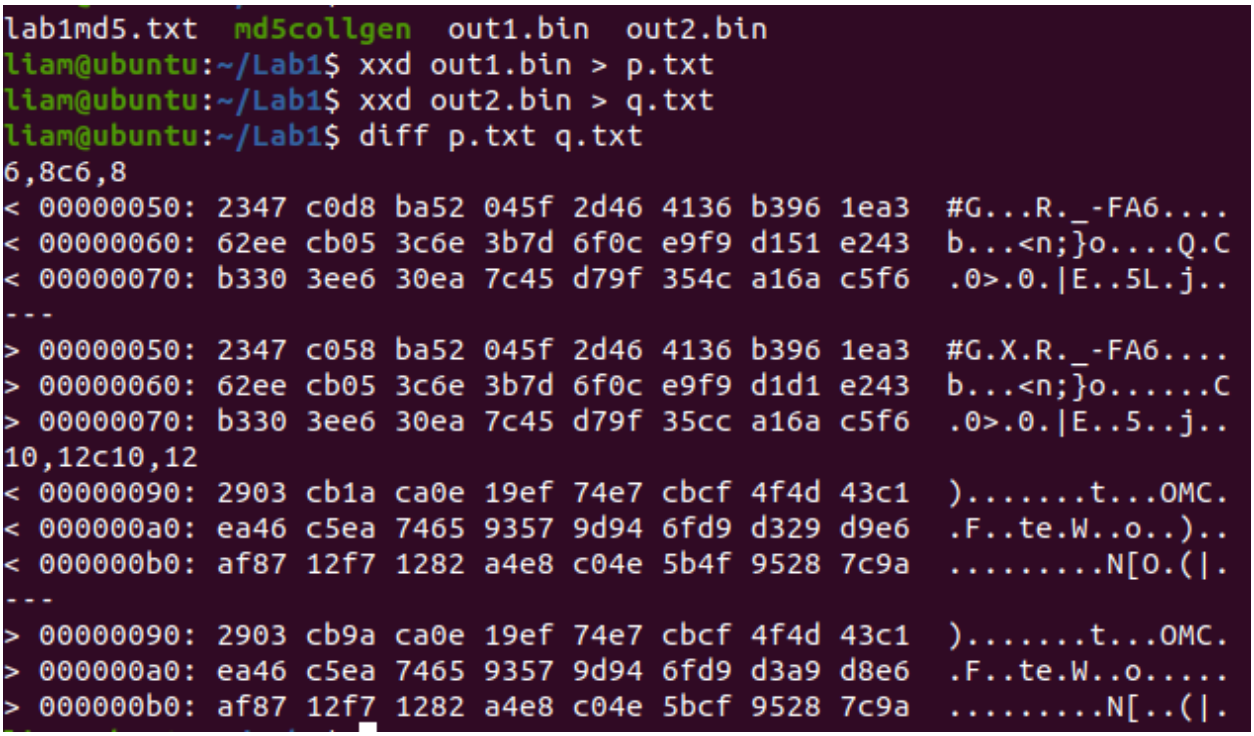
Are the data (128 bytes) generated by md5collgen completely different for the two output files? Please identify all the bytes that are different.

A small portion of the bytes are different and scattered throughout.

Lines affected in p.txt are denoted by <

Lines affected in q.txt are denoted by >

The diff command outputs a set of instructions for how to change one file to make it identical to the second file [1]. In this case we have 6,8c6,8 and 10,12c10,12 where c means change. Thus, this output can be interpreted as lines 6-8 in the first file (p.txt) need to be changed to lines 6-8 in the second file (q.txt) and lines 10-12 in p.txt need to be changed to lines 10-12 in q.txt.



Task 2: Further Understanding MD5's Property

If two inputs to the MD5 algorithm create a collision, then adding the same suffix (T) to them will result in a collision as well. To demonstrate this, I created two new files f1.txt and f2.txt with the same contents inside each file. To ensure a collision occurred I ran a checksum on both files and sure enough the MD5 hashes for each file are the same. This satisfies the initial part of the task in that *MD5(M_1) = MD5 (M_2)*.

```
liam@ubuntu:~/Lab1$ md5sum f1.txt
f30834534bedae23808b9a4d2b1464e9  f1.txt
liam@ubuntu:~/Lab1$ md5sum f2.txt
f30834534bedae23808b9a4d2b1464e9  f2.txt
```

The next part of the task is to satisfy the following condition: *for any input T, MD5(M_1 // T) = MD5(M_2 //T), where // means concatenate*. To do this I created a new text file and concatenate it to both f1.txt and f2.txt. Finally, I re-run the md5 check sum on the concatenated files and it again returns two identical MD5 Hashes.

```
liam@ubuntu:~/Lab1$ echo "suffix" >> suffix.txt
```

```
liam@ubuntu:~/Lab1$ cat f1.txt suffix.txt  > f3.txt
liam@ubuntu:~/Lab1$ cat f2.txt suffix.txt > f4.txt
liam@ubuntu:~/Lab1$ ls
f1.txt  f3.txt  lab1md5.txt  out1.bin  p.txt  suffix.txt
f2.txt  f4.txt  md5collgen   out2.bin  q.txt
liam@ubuntu:~/Lab1$ md5sum f3.txt
1b798f4c1d8d556a6dc2cfc7b7187a63  f3.txt
liam@ubuntu:~/Lab1$ md5sum f4.txt
1b798f4c1d8d556a6dc2cfc7b7187a63  f4.txt
liam@ubuntu:~/Lab1$
```

Thus, we know that if two objects have the same hash, the concatenation of data onto both objects will result in them having the same hash as well.

Task 3: Generating Two Executable Files with the Same MD5 Hash

Below is a sample executable C program. The array contents are what will vary in this program.

```c
#include <stdio.h>
unsigned char xyz[200] = {
/* The contents of this array will vary */
};
int main()
{
  int i;
  for (i=0; i<200; i++){
  printf("%x", xyz[i]);
  }
  printf("\n");
}
```

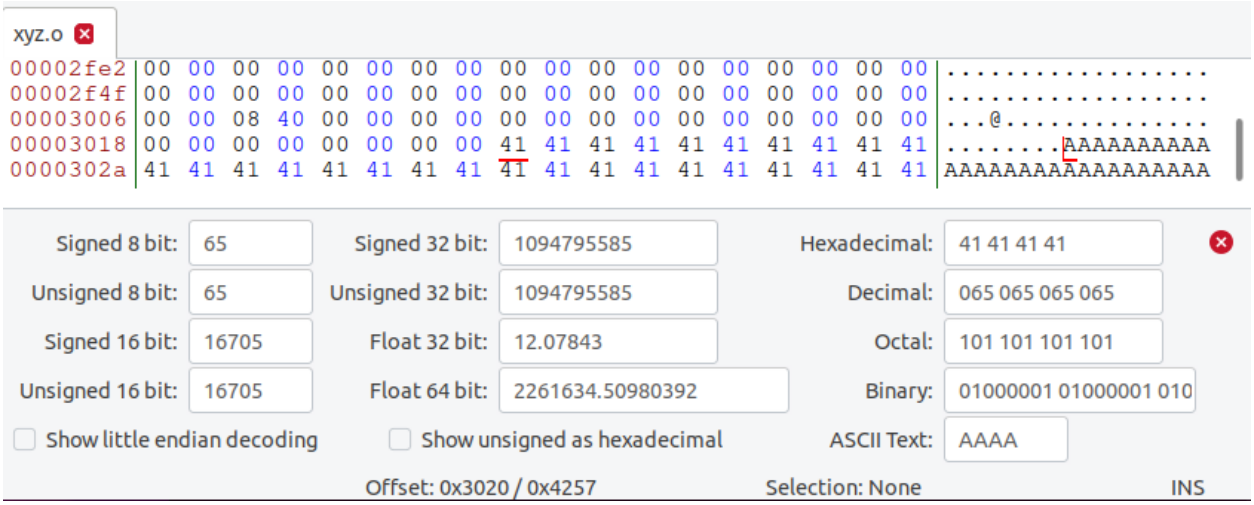
I had previously typed out 64 As in task 1 and then double checked the length with the *ls -l* command and option. However, I was not about to type out 200 A's and since making files of some size with arbitrary content seemed to be a reoccurring theme, I searched for a method and found one that would streamline this process for me [2]. These 200 A's will be inserted into the array so when I dump the hex I can easily find where it starts.

```
liam@ubuntu:~/Lab1$ echo -e "print('0x41,'*199)" | python3 > fileofA
```

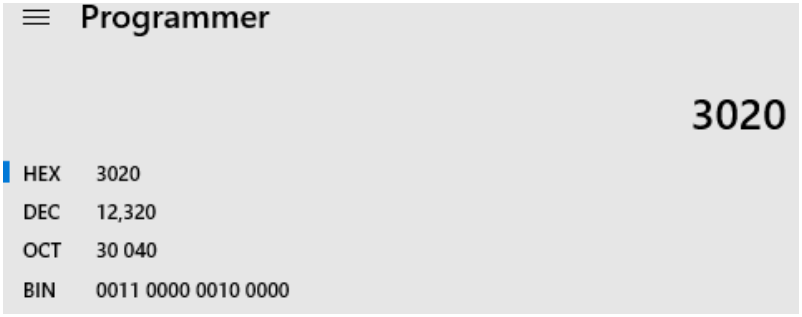
Then copying all these contents into the array in the provided c program, I compiled with gcc and created an output file xyz.o.

```
liam@ubuntu:~/Lab1$ gcc xyz.c -o xyz.o
```

Skimming the hex, it was easy to find where the array started, and the offset.



The offset is given in hex, converting to decimal it is 12,320. Dividing by 64 tells us if it is a multiple of 64, and it is not so I must jump to the next multiple which is at 0x3040 or decimal 12352.



The prefix required will contain everything before the array and a portion of the array, to ensure our prefix remains a multiple of 64 I just jumped to the next offset that was a multiple of 64.

The prefix is followed by a 128-bytegap (where a p or q value goes) and then the suffix. So, our prefix +

128 + 1 is the offset for suffix. *12352 + 128 + 1 = 12,481*.

Now I want to use these offsets to create the prefixes and suffixes.

```
liam@ubuntu:~/Lab1$ head -c 12352 xyz.o > prefix
liam@ubuntu:~/Lab1$ tail -c +12481 xyz.o > suffix
liam@ubuntu:~/Lab1$
```

Generate collisions with the prefix.

```
liam@ubuntu:~/Lab1$ ./md5collgen -p prefix -o p.bin q.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'p.bin' and 'q.bin'
Using prefixfile: 'prefix'
Using initial value: b9a6f7bbceb320b0dc6128ad2226f1c2

Generating first block: ..............................
Generating second block: S00.....
Running time: 22.3524 s
liam@ubuntu:~/Lab1$
```

I then double check that they do produce the same hash value. They do which means MD5 (prefix || P) = MD5 (prefix || Q) is satisfied.

```
liam@ubuntu:~/Lab1$ md5sum p.bin
3f16da364c59d295e1408081bccdac20  p.bin
liam@ubuntu:~/Lab1$ md5sum q.bin
3f16da364c59d295e1408081bccdac20  q.bin
liam@ubuntu:~/Lab1$
```

```
liam@ubuntu:~/Lab1$ cat prefix p > prefixP
liam@ubuntu:~/Lab1$ cat prefix q > prefixQ
liam@ubuntu:~/Lab1$ md5sum prefixP
923999c9562b2594196ca1cce8001358  prefixP
liam@ubuntu:~/Lab1$ md5sum prefixQ
923999c9562b2594196ca1cce8001358  prefixQ
liam@ubuntu:~/Lab1$
```

Concatenating the suffix to the previous part we end up at the final step (prefix || P || suffix) and (prefix || Q || suffix). Performing a checksum on the two files I get that they are the same meaning the final part of this task has been satisfied. MD5 (prefix || P || suffix) = MD5 (prefix || Q || suffix).

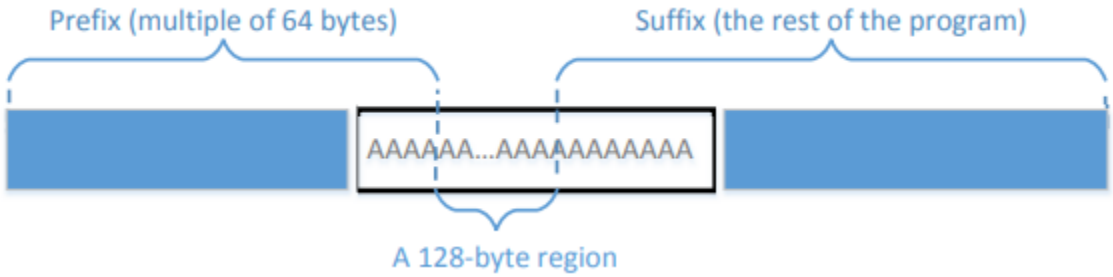
```
liam@ubuntu:~/Lab1$ cat p.bin suffix > prefixPsuffix
liam@ubuntu:~/Lab1$ cat q.bin suffix > prefixQsuffix
liam@ubuntu:~/Lab1$ md5sum prefixPsuffix
e335006bee9e7df95f0acb6562b28954  prefixPsuffix
liam@ubuntu:~/Lab1$ md5sum prefixQsuffix
e335006bee9e7df95f0acb6562b28954  prefixQsuffix
liam@ubuntu:~/Lab1$
```

The final step is to ensure that the programs do in fact produce different lists. I chmod them both to be executable and it produces some output (the lists). In that output it is not easy to see the difference, so I rerun the programs and put the input in a file and run diff on those two files. Diff determined that they are different arrays. Thus, we have two programs that produce different lists and have the same checksum.

```
liam@ubuntu:~/Lab1$ chmod +x prefixPsuffix
liam@ubuntu:~/Lab1$ chmod +x prefixQsuffix
liam@ubuntu:~/Lab1$ ./prefixPsuffix
4141414141414141414141414141414141414141414141414141414141414141c30948670267ac653ddf11caafd82a36dc654deb9c6a23d6637b56b8fc0b270ab2e3f2e6498f6f5ad7bb1baa97527b0
fb6842f3dd1cdc338aa3bfc34dd6b6d3805038d037b12d6a9d713779fccaf252ff874e160cd57f5c0d9607d22fb4e9fc22e437355a18bd42fcf1d76e4dfffda33c72b6e43231d7668553d9524141414
1414141414141414141414141414141414141414141414141414141414141414141414141
liam@ubuntu:~/Lab1$ ./prefixQsuffix
4141414141414141414141414141414141414141414141414141414141414141c30948670267ac653ddf11caafd82a36dc65cdeb9c6a23d6637b56b8fc0b270ab2e3f2e6498f6f5ad7bb1b2aa7527b0
fb6842f3dd1cdc338a23bfc34dd6b6d3805038d037b12d6a9d713779fccaf25aff874e160cd57f5c0d9607d22fb4e9fc22e437355a18bd42fcf9d75e4dfffda33c72b6e43231d7e68553d9524141414
1414141414141414141414141414141414141414141414141414141414141414141414141
liam@ubuntu:~/Lab1$ ./prefixPsuffix > imp
liam@ubuntu:~/Lab1$ ./prefixQsuffix > imq
liam@ubuntu:~/Lab1$ diff imp imq
1c1
< 4141414141414141414141414141414141414141414141414141414141414141c30948670267ac653ddf11caafd82a36dc654deb9c6a23d6637b56b8fc0b270ab2e3f2e6498f6f5ad7bb1baa97527
b0fb6842f3dd1cdc338aa3bfc34dd6b6d3805038d037b12d6a9d713779fccaf252ff874e160cd57f5c0d9607d22fb4e9fc22e437355a18bd42fcf1d76e4dfffda33c72b6e43231d7668553d95241414
1414141414141414141414141414141414141414141414141414141414141414141414141
---
> 4141414141414141414141414141414141414141414141414141414141414141c30948670267ac653ddf11caafd82a36dc65cdeb9c6a23d6637b56b8fc0b270ab2e3f2e6498f6f5ad7bb1b2aa7527
b0fb6842f3dd1cdc338a23bfc34dd6b6d3805038d037b12d6a9d713779fccaf25aff874e160cd57f5c0d9607d22fb4e9fc22e437355a18bd42fcf9d75e4dfffda33c72b6e43231d7e68553d95241414
1414141414141414141414141414141414141414141414141414141414141414141414141
```

To elaborate on the above results what we have done is taken a program, dissected its hex to find the offset where the array starts and then found some offset within the array that was a multiple of 64. It must be a multiple of 64 so that no padding is added. Finding this offset leaves us with three keys pieces:

- **Prefix:** some multiple of 64 that goes from the beginning of the program to a point within the array.
- **128-byte region:** A portion within the array after the prefix and before the suffix that contains either a P value or a Q value. These P and Q values are like the magic values that will allow us to have two different programs with the same checksum.
- **Suffix:** Everything in the program after the 128-byte region.



[3]

Task 4: Making the Two Programs Behave Differently

The idea here is that the certificate check is only concerned with the hash value of the program and not the code itself, so in code I can have malicious and benign routes of execution. For this lab this requires us to create a collision. The execution route is determined by whether two lists are identical, if they are identical, execute benign code, if they aren't do something malicious. However, in general this would cause a change in the program data that would change the hash value of it, nullifying the certificate. If we can change the contents of the list without changing the hash, we can circumvent this. This is where the collision comes in. By going from the start of the program to the start of the first array (or where inside it is the first multiple of 64) we can use MD5collgen to create a prefix that can be used to generate a collision, that is, the start of the program with a differing 128-byte ending (p & q). Then we append a suffix (the rest of the program after our prefix). If we copy the 128-byte P value and paste it into the same part in the second array as the first, we have a program that will run the benign code and produce a hash value of x. This is what we would send in to get certified. Then we can substitute the 128-byte p value in one of the arrays with the 128-byte q value. The contents will be different, but the hash will still be x, and now the execution will be malicious. I started by creating a program that would do something benign when the contents of two arrays are the same and then do something else when they are not the same, then complied and ran it to make sure it worked.

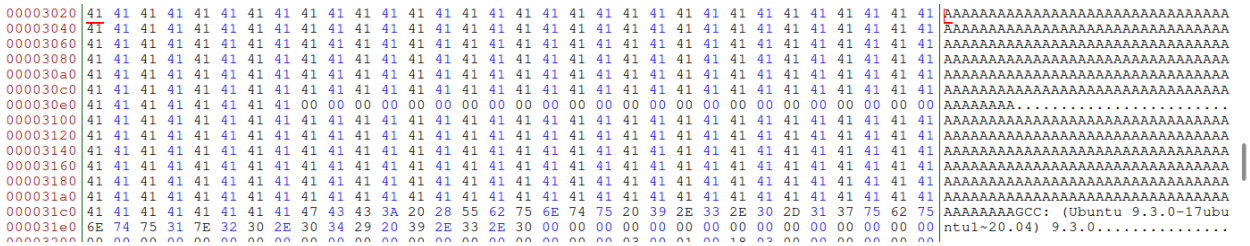
```c
#include <stdio.h>

unsigned char x[200] = {
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41
};
```

```c
unsigned char y[200] = {
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41
};

int main()
{
  int i;

  for (i = 0; i<200; i++){
    if (x[i] != y[i]){
      printf("malicious\n");
      break;
    }else if(i==199){
      printf("benign\n");
    }
  }
  return 1;
}
```

On the first run I changed a single byte in list x to make sure the cases worked. On the second run I changed the byte back 0x41.

```
liam@ubuntu:~/Lab1$ atom task4prog.c
liam@ubuntu:~/Lab1$ gcc task4prog.c -o task4prog.o
liam@ubuntu:~/Lab1$ ./task4prog.o
malicious
liam@ubuntu:~/Lab1$ atom task4prog.c
liam@ubuntu:~/Lab1$ gcc task4prog.c -o task4prog.o
liam@ubuntu:~/Lab1$ ./task4prog.o
benign
```

Next, I dumped the program and started looking for the offset of the first list. From the previous example I know that an offset of 0x3020 is 12,320 in decimal and further that for my prefix I want to remain a multiple of 64. The next offset that is a multiple of 64 is 0x3040, which in decimal is 12352. Thus, my prefix ends at byte 12352. To get my suffix I need to jump 128-bytes + 1, making the beginning of my suffix 12481.




```
liam@ubuntu:~/Lab1/lab1t4$ head -c 12352 task4prog.o > prefix
liam@ubuntu:~/Lab1/lab1t4$ tail -c +12481 task4prog.o > suffix
liam@ubuntu:~/Lab1/lab1t4$
```

Using the MD5 collision generator, I get a p and q value based on my prefix.

```
liam@ubuntu:~/Lab1/lab1t4$ ./md5collgen -p prefix -o prefixP.bin prefixQ.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'prefixP.bin' and 'prefixQ.bin'
Using prefixfile: 'prefix'
Using initial value: 29b7e5ea57f3cc3a494c13646ac6e04e

Generating first block: ............
Generating second block: S00.........
Running time: 10.252 s
liam@ubuntu:~/Lab1/lab1t4$
```

Confirming they are different with the same checksum.

```
liam@ubuntu:~/Lab1/lab1t4$ cat prefixP.bin suffix > p_injection
liam@ubuntu:~/Lab1/lab1t4$ cat prefixQ.bin suffix > q_injection
liam@ubuntu:~/Lab1/lab1t4$ diff prefixP.bin prefixQ.bin
Binary files prefixP.bin and prefixQ.bin differ
liam@ubuntu:~/Lab1/lab1t4$ md5sum prefixP.bin
bd93f80081f77a523c3ae24b609d033a  prefixP.bin
liam@ubuntu:~/Lab1/lab1t4$ md5sum prefixQ.bin
bd93f80081f77a523c3ae24b609d033a  prefixQ.bin
liam@ubuntu:~/Lab1/lab1t4$
```

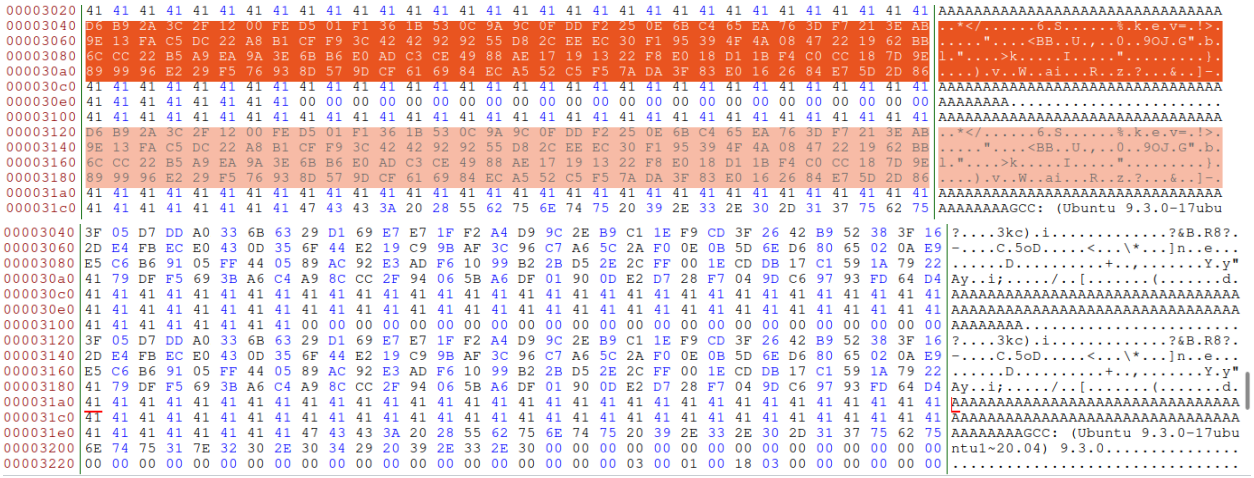
p_injection with the prefix + p value + suffix looks like this, where the first array has the injected p value injected at 0x3040 and the second array is untouched for now. If I were to chmod p_injection and execute it, it would run the malicious code at this point in time because the arrays are different.

```
00003020 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00003040 d6 b9 2a 3c 2f 12 00 fe d5 01 f1 36 1b 53 0c 9a 9c 0f dd f2 25 0e 6b c4 65 ea 76 3d f7 21 3e ab ..*</.....6.S.....%.k.e.v=.!>.
00003060 9e 13 fa c5 dc 22 a8 b1 cf f9 3c 42 42 92 92 55 d8 2c ee ec 30 f1 95 39 4f 4a 08 47 22 19 62 bb .....".....<BB..U.,..0..9OJ.G".b.
00003080 6c cc 22 b5 a9 ea 9a 3e 6b b6 e0 ad c3 ce 49 88 ae 17 19 13 22 f8 e0 18 d1 1b f4 c0 cc 18 7d 9e l."....>k.....I....."........}.
000030a0 89 99 96 e2 29 f5 76 93 8d 57 9d cf 61 69 84 ec a5 52 c5 f5 7a da 3f 83 e0 16 26 84 e7 5d 2d 86 ....).v..W..ai...R..z.?...&..]-.
000030c0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
000030e0 41 41 41 41 41 41 41 41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 AAAAAAAA........................
00003100 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00003120 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00003140 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00003160 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00003180 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
000031a0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
000031c0 41 41 41 41 41 41 41 47 43 43 3A 20 28 55 62 75 6e 74 75 20 39 2e 33 2e 30 2d 31 37 75 62 75 AAAAAAAGCC: (Ubuntu 9.3.0-17ubu
```

q_injection with the prefix + q value + suffix looks like this, where the first array has the injected q value at its start and the second array is untouched for now. Similar to before, running this right now would run the malicious code.

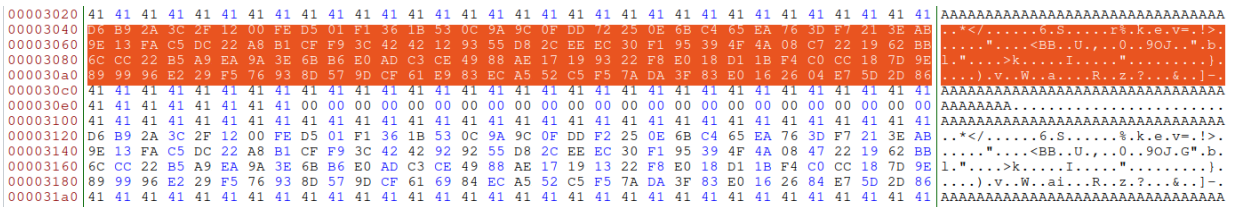
```
00003040 3f 05 d7 dd a0 33 6b 63 29 d1 69 e7 e7 1f f2 a4 d9 9c 2e 39 c1 1e f9 cd 3f 26 42 b9 52 38 3f 16 ?....3kc).i........9....?&B.R8?.
00003060 2d e4 fb ec e0 43 0d 35 6f 44 e2 19 c9 1b b0 3c 96 c7 a6 5c 2a f0 0e 0b 5d 6e d6 00 65 02 0a e9 -....C.5oD.....<...\*...]n..e...
00003080 e5 c6 b6 91 05 ff 44 05 89 ac 92 e3 ad f6 10 99 b2 2b d5 ae 2c ff 00 1e cd db 17 c1 59 1a 79 22 ......D.........+..,........Y.y"
000030a0 41 79 df f5 69 3b a6 c4 a9 8c cc 2f 94 86 5a a6 df 01 90 0d e2 d7 28 f7 04 9d c6 17 93 fd 64 d4 Ay..i;...../..Z.......(.......d.
000030c0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
000030e0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00003100 41 41 41 41 41 41 41 41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 AAAAAAAA........................
00003120 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00003140 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00003160 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00003180 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
000031a0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
000031c0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
000031e0 41 41 41 41 41 41 41 41 47 43 43 3A 20 28 55 62 75 6e 74 75 20 39 2e 33 2e 30 2d 31 37 75 62 75 AAAAAAAAGCC: (Ubuntu 9.3.0-17ubu
00003200 6e 74 75 31 7e 32 30 2e 30 34 29 20 39 2e 33 2e 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ntu1~20.04) 9.3.0...............
00003220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 01 00 18 03 00 00 00 00 00 00
```

Copying the p portion of code in p_injection and inserting it into the same position in the second array. The code should see the arrays as identical and run the benign code after changing the permissions of p_injection.




```
liam@ubuntu:~/Lab1/lab1t4$ chmod +x p_injection
liam@ubuntu:~/Lab1/lab1t4$ ./p_injection
benign
liam@ubuntu:~/Lab1/lab1t4$ md5sum p_injection
e5b2bdc6875e14b3f145cc1fb9285bae  p_injection
liam@ubuntu:~/Lab1/lab1t4$
```

Now in-order to make the malicious code run we want to inject the p value into one array and the q value into the other at the same positions. In doing this the code will identify the characters in the array as different and run the alternative malicious code, but because of the collision generated by md5collgen the hash will be identical. The highlighted potion is now the q portion, and second array contains the p portion.




```
liam@ubuntu:~/Lab1/lab1t4$ chmod +x q_injection
liam@ubuntu:~/Lab1/lab1t4$ ./q_injection
malicious
liam@ubuntu:~/Lab1/lab1t4$
```

Interesting bumps and hurdles I ran into:

- Apparently, bless doesn't have a tmp file to save to by default in Ubuntu when modifying hex directly, so I had to go into the .config/preferences.xml of bless and add a file path for it to save to so that I could do task 4 the way I did, where I copy bytes from one array and paste them into the other and then save it.
    - It looks something like this <pref name="ByteBuffer.TempDir"><</pref>
    - between the two red arrows I added a file and it fixed it.

Citations

https://www.computerhope.com/unix/udiff.htm [1]

https://stackoverflow.com/questions/2043453/executing-multi-line-statements-in-the-one-line-command-line [2]

SEED Project (seedsecuritylabs.org) [3]